Evaluation of OpenMP Task Scheduling Algorithms for Large NUMA Architectures

Jérôme Clet-Ortega, Patrick Carribault, and Marc Pérache

CEA, DAM, DIF F-91297, Arpajon, France
{jerome.clet-ortega,patrick.carribault,marc.perache}@cea.fr

Abstract. Current generation of high performance computing platforms tends to hold a large number of cores. Therefore applications have to expose a fine-grain parallelism to be more efficient. Since version 3.0, the OpenMP standard proposes a way to express such parallelism through tasks. Because the task scheduling strategy is implementation defined, each runtime can have a different behavior and efficiency. Notwithstanding, the hierarchical characteristic of current parallel computing systems is rarely considered. This might come down to a loss of performance on large multicore NUMA systems. This paper studies multiple task scheduling algorithms with a configurable scheduler. It relies on a topology-aware tree-based representation of the computing platform to orchestrate the execution and the load-balacing of OpenMP tasks. High-end users can select the task-list granularity according to the tree structure and choose the most convenient work-stealing strategy. One of these strategies takes into account data locality with the help of the hierarchical view. It performs well with unbalanced codes, from BOTS benchmarks, in comparison to Intel and GNU OpenMP runtimes on 16-core and 128-core systems.

1 Introduction

Conceiving parallel algorithms is getting more and more intricate in accordance with the evolution of computer architectures. Multi-core and many-core systems are widespread in the high performance computing landscape. The number of computing units per node massively increase and the future processor design announced by constructors, for example Intel *Xeon Phi* [6], continues this upward trend. In order to help the parallel application programmer in getting the best performance from the hardware, work has been conducted to integrate inside programming model implementations several mechanisms [1,2,3] that take into account the memory hierarchy of the underlying node. The programming models themselves evolve to offer features fitted with current processors structure. One could cite the adjonction of task parallelism to the OpenMP *de facto* standard [4] (in the 3.0 version) that allow the programmer to express a fine-grained parallelism. Currently, most of the OpenMP implementations support task programming, like in GNU OpenMP [5] or Intel OpenMP [7]. Thus, each of them relies on a particular task management system which directly affects the application performance, according to the system architecture. Indeed,

F. Silva et al. (Eds.): Euro-Par 2014, LNCS 8632, pp. 596–607, 2014.
© Springer International Publishing Switzerland 2014

C. Terboven and al. [8] point out that the topology needs to be taken into account, especially on NUMA architectures. In this article, we propose to draw a list of parameters that control task scheduling and evaluate the different configurations with representative applications over a highly hierarchical system. In this way, we characterize the models of task-based applications and we map each category to the right OpenMP task scheduling configuration. We implemented our work inside the MPC framework [9,21], which now includes an OpenMP 3.0 implementation.

This paper is structured as follows. The next section presents some related work on task scheduling. Section 3 introduces the task scheduling parameters (topology-driven task-list granularity and work-stealing policy) and the design of our customizable OpenMP task runtime. Evaluation results of the different combinations provided by our proposal are presented in section 4. The last section sums up these results and deals with future work.

2 Related Work

To extract the most relevant runtime parameters shaping the performance of task scheduling, we first focus on several up-to-date task implementations. One of the most common impementation is the GNU OpenMP runtime (libGOMP) embedded within the GNU compiler collection. For the task scheduling, it allocates a single list per OpenMP team. Each access to this global list (creation of a deferred task or regular task scheduling) is synchronized with a mutex. Therefore, when a large amount of threads execute tasks, it quickly becomes a memory bottleneck.

The other major implementation, Intel OpenMP *Runtime Library*, attaches a task list to each OpenMP thread of every team. Each time a thread creates a task that will not be executed immediatly, that task is placed inside the thread's deque (double-ended queue). A random stealing strategy between deques is set up for load balancing purpose. This implementation design is used by several other approaches: OpenUH, X-KAAPI and OMPi. The first one, OpenUH, is a branch of the Open64 compiler suite providing OpenMP tasking feature [10]. Inspired by the Cilk scheduler [11], the tasks are created in a breadth-first way and executed in depth-first manner. It also uses a cut-off (on the number of total tasks created and the depth in the task graph) to avoid task overload which could happen with recursive algorithm like Fibonacci sequence computing. Based on the runtime system for data-flow parallel applications X-KAAPI [15], LibKOMP [16] implements the OpenMP task model. Load balancing is realized with a work-stealing technic, inspired by Cilk. They also propose several extensions to the standard to deal with task data dependencies now present in OpenMP 4.0. Nevertheless, in this article we do not deal with this part of the standard which add some interesting constraints to task scheduling. Finally, OMPi [17] is an OpenMP 3.0 infrastructure for C language, including a source-to-source compiler and a runtime library. In order to implement a breadth-first algorithm, it uses a circular deque per thread to manage deferred tasks. Load

balancing is managed with a lock-free work-stealing algorithm where each thread thief traverses other thread queues.

The ROSE compiler infrastructure [12] exposes an approach beyond per-thread list and random stealing. Its OpenMP task scheduling implementation [13] relies on *Qthreads* user-level thread runtime library [14]. It targets multi-core systems through a hierachical scheduling strategy. They point out a strategy which regroups threads in *shepherds* using one LIFO task queue per *shepherd*. A work-stealing mechanism between *shepherds* maintains the load balancing.

Finally, other approaches deal with task list placement and topology-driven work-stealing. Thus, the ForestGOMP software [2] proposes work-stealing technics according to the memory hierarchy. However this takes place when dealing with nested parallelism and the stealing objects are threads and their data. Beside OpenMP runtimes, StarPU [18] is well-known in HPC domain and is based on data-flow dependencies to schedule tasks. A set of parameters is available to define the kind of task list (FIFO, LIFO, deque) and its granularity (one per thread or a single global one).

From this overview of task schedulers and their internal mechanisms, we decide to build an environment in which we could play with different parameters like the granularity of task lists and the work stealing policy.

3 Task Scheduling Control

OpenMP tasking support implies for a runtime developer to take some significant decisions concerning the implementation, especially the type of datastructure for task management (list, stack, deque, ...) which bears directly on application performance. Indeed, the task paradigm often leads OpenMP applications to generate a large amount of tasks. That means the runtime has to minimize the overhead of the numerous operations linked to task managing: creation, browsing, sharing and/or stealing, ... Additionally, the runtime has to control the load balancing, through a work stealing mechanism in most cases.

This section details some relevant runtime parameters shaping the performance of task scheduling and the design of our customizable OpenMP task scheduling engine. It gives us the possibility to evaluate multiple configurations inspired from the existing runtimes described in the previous section. The next section will illustrate such configurations with experimental results.

3.1 Task List Granularity

The omnipresence of multicore architectures in the High Performance Computing landscape constrains the application and runtime developers to take care of the underlying hardware topology. Some work on OpenMP task scheduling [2,13,8] has shown that the difficulty comes from mapping of task execution scheme with the memory hierarchy of the system. The first relevant parameter is therefore the placement of task lists according to the hardware topology.

We rely on the HWLOC software package [19], used in several parallel runtimes and MPI implementations, to discover the entire topological structure of the system. It allows us to build a topology tree for OpenMP teams based on a restricted view of the original one: it ignores the levels that do not bring any structure information (one-to-one links). Thus this tree holds all the groups of threads defined according to the memory hierarchy, as in the ForestGOMP runtime system with its hierarchical scheduler. For example, on a dual-socket eight-core Intel *Sandy Bridge EP*, each core owns a L1 and a L2 cache memory and the L3 cache is shared between the eight cores of a socket. Each socket is linked with a memory bank, constituting a NUMA node. Thus, there are eight levels but only three are relevant in the hierarchy structure. The first one concerns the processing units, the cores, the L1 and L2 cache memories. The second one regroups the L3 cache and the NUMA node levels. The last one is the whole machine.

With this representation, the user can decide at which level the task lists are allocated and accessed by the OpenMP threads. This parameter allows to check the impact of access contention to the list(s) and data locality, noticed as a main challenge [8]. Thus we extend the *shepherds* concept of S.L. Olivier and al. [13] to the whole hierarchical levels of a computing node.

3.2 Stealing Strategies

The second parameter identified in Section 2 is the stealing strategy. Indeed, since the development of *Cilk* [11], the work-stealing algorithm is the most studied one for dynamic load balancing purpose. It provides pretty good performance on average and is implemented inside a large majority of task schedulers. When it comes to starvation for a thread, it becomes a thief looking for work inside other task pools. Most of the time the victim is randomly chosen: it often avoids contention for multiple thieves at the same time and is a quick decision algorithm which matters at such critical point. However, this strategy does not take into account the memory position of the stolen data according to the binding of the thief thread. On a SMT system or a small scale system, that does not really matter. When executing this algorithm on a large multiprocessor and multicore machine, the impact over the performance may not be negligible anymore. So, for this second parameter, we designed several policies for the selection of the victim. This can be divided into three categories.

The first category of stealing strategy is based on random-victim choice. We can extract two approaches: *Random* and *Random Order*. The first one looks for a randomly-chosen victim and the second one generates a random-ordered sequence of all task lists to look for. The second category exploits the hierarchical aspect of the underlying topology. The *Hierarchical* strategy starts from the closest list in the hierarchical order determined with the physical architecture structure to choose a list to steal. For example, in case of one task list per thread, a thread whose list is empty will start to steal a task from the lists of threads running on the cores of the same processor, before looking further. Another hierarchical approach is called *Round Robin*: it browses the lists for a task to

steal according to a static global topology ordering. A starving thread would look inside the first neighbor of its own list, then the second, and so on. The first neighbor is likely to be close according to the hardware topology but may not be the closest one. The advantage of this approach is the low overhead to choose the victim. The final category is based on a per-list statistic: the *Producer* algorithm selects the list which contains the largest number of tasks enqueued and the *Producer Order* strategy builds a sequence of task lists according to this indicator.

3.3 Implementation

To evaluate the impact of the 2 parameters (task-list granularity and stealing strategy), we implemented a customizable OpenMP task scheduling engine. This work has been realized inside the unified parallel-runtime framework called MPC [9], by extending the current OpenMP library to conform to the OpenMP 3.0 standard. The MPCOMP library uses a tree-based representation of the computing platform to schedule OpenMP threads. We relied on this tree to develop our task scheduling engine. Task lists are allocated on the appropriate NUMA node thanks to this structure and stealing strategies functions use it to retrieve tasks to execute. The compilation step, which turns OpenMP directives into runtime calls, is performed by a patched version of the GNU C compiler. Finally, we could control at user level the behavior of the scheduler by specifying the number of task lists and choosing the work stealing policy.

4 Evaluation

This section reports the results of our experiments on the BOTS benchmarks suite to evaluate our different strategies in comparison to two other OpenMP implementations: the Intel OpenMP *Runtime Library* coming with version 13.1.3 of the Intel C compiler and the GNU OpenMP library with the version 4.7 of the GNU C compiler.

4.1 Experiments Platforms

Our experiments were conducted on a 128-core node of the *Curie* supercomputer (GENCI) and composed of 16 eight-core Intel *Nehalem-EX* processors at 2.27 GHz and associated to 512 GB of memory (32 GB per NUMA node). This structure comes from the association of 4 motherboards inter-connected through a Bull network. Each processor exposes three levels of cache memory: 32 KB of L1 and 256 KB of L2 cache owned by each core and 24 MB of L3 shared by eight cores.

In our implementation in MPC, the hierarchical representation of the system, used for OpenMP thread scheduling, is a four-level tree. The root (level 0) corresponds to the whole computational node, the next level (1) to the motherboard, the next one (level 2) to the socket and L3 cache memory and the leaves

(level 3) represent the cores with their L1 and L2 cache memories. That means the task-list granularity (first parameter) could be defined among four values: a single one for whole system (1 list), one list per motherboard (4 lists), one list per socket (16 lists) and one list per core (128 lists).

4.2 Results

The results come from the execution of the *Barcelona OpenMP Tasks Suite* (BOTS). These benchmarks, inspired from real-life applications, evaluate the performance of OpenMP tasks runtimes. Several versions of each benchmark are available and described in [20]. The next array presents the main characteristics of those benchmarks we took interested in. Thus, some kernels use a single thread to produce all the tasks (with a *single* construct) whereas for others, all threads generate a certain amount of tasks. Three of them use tasks that may generate tasks (nesting). Finally, only the SparseLU benchmark exhibits an irregular parallelism.

Application	Creation pattern	Task type	Load-balancing	#Tasks
Alignment	Single & Multiple	Final	Regular	$\approx 50K$
FFT	Multiple	Nested	Regular	$\approx 10M$
Fibonacci	Multiple	Nested	Regular	$\approx 860M$
Sort	Single	Nested	Regular	$\approx 2M$
Sparse LU	Single & Multiple	Final	Irregular	$\approx 40K$

One has to specify that each compiler provides different sequential performance. For example, the modified version of libGOMP used for MPCOMP is 4.4 and the version used for comparisons is 4.7. Because of this difference, the MPCOMP version is on average 30-35 % less effective than libGOMP on serial execution of these benchmarks. There is a similar impact with the Intel compiler which generates most of the time a faster serial code on the target Intel-based architecture.

4.3 Alignment

Alignment is an application where the data locality really matters. Indeed, the quantity of write operations to non-private memory is very low as presented in [20]. The great majority of stores are to the private memory of the task.

Figures 1 and 2 present the speed-up obtained on 128 cores with 128 OpenMP threads for two versions: multiple producers and single producer. Regarding the first parameter, the task scheduling with a single global list, as for libGOMP, is the worst solution whereas giving one list per thread, like Intel OpenMP, seems to be more efficient. For the second parameter (work-stealing approaches), all strategies deliver more or less the same performance. Gaps between strategies execution time are just a bit less perceptible with the *single* construct version.

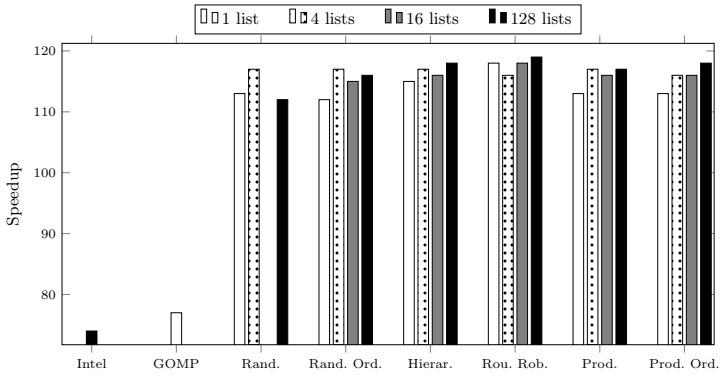

Fig. 1. Speedup for *Alignment* multiple producer benchmark with 128 threads

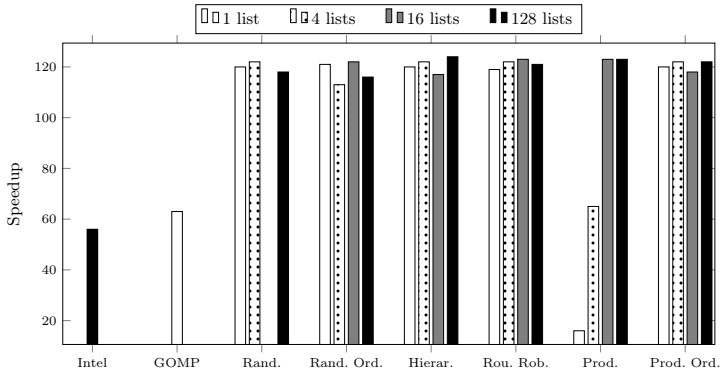

Fig. 2. Speedup for *Alignment* single producer benchmark with 128 threads

Overall, a hierarchical strategy seem to give the best performance (either hierarchical or round robin).

Of interest, for both single and multiple producers versions, MPCOMP performs worse than Intel OpenMP and even than libGOMP in serial: the run takes 2,791 seconds for MPCOMP whereas Intel OpenMP performs it in 1,312 seconds and libGOMP in 2,105 seconds. The performance follow the same trend with 8, 16, 32, 64 threads. When it reaches the number of 128 threads, MPCOMP distinguishes itself and outperforms libGOMP. For a well balanced code, like the multiple producer version, Intel OpenMP still reaches better performance. However with the *single* construct, the benchmark is unbalanced and MPCOMP outperforms Intel OpenMP. The best results comes from the *Hierarchical* stealing policy, always with one task list per thread. This is consistent with the fact that hierarchical strategy favors local data accesses for the benefit of this benchmark.

4.4 FFT

The *FFT* benchmark computes the *Fast Fourier Transform* of a vector of n complex values. Intel OpenMP gave the best results in this benchmark even if the speedup is not really significant (10 on a 128 cores). For its own part, libGOMP does not scale and performs really worse than Intel OpenMP (approximately 50 times longer). Among several policies given by MPCOMP, the ones using a single list per core show the best performance, leading to an execution time 10 times faster but still 5 times longer than Intel OpenMP. Moreover all other policies deliver a consequent deterioration of the execution time. One may notice that, with 32 threads, the best approach uses one list per socket and the producer strategy for work-stealing.

There are a really large number of tasks to manage for the runtime. In order to limit the impact of the overhead for managing so many elements, GCC use a static threshold on the number of tasks generated at one time.[1] We also choose this solution. However the parallelism of the application is limited by this threshold. This may explain the performance difference between the Intel runtime and the libGOMP and MPCOMP implementations.

4.5 Fibonacci

This application benchmark generates a large number of fine grained OpenMP task to compute the n^{th} Fibonacci number thanks to a recursive algorithm. For our run we use the parameter $n = 42$.

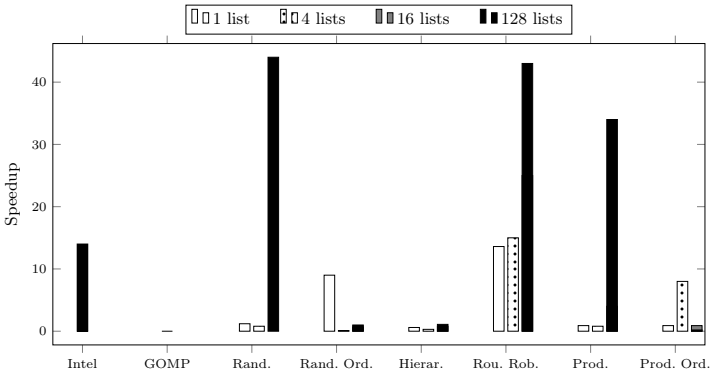

Fig. 3. Speedup for *Fibonacci* benchmark with 128 threads

Figure 3 shows the results for this benchmark. Among all strategies, only the ones with a low overhead like *Random*, *Round Robin* and *Producer*, deliver a good speedup, even better than Intel OpenMP. Indeed, in this *Fibonacci*

[1] Actually, the threshold is defined to 64 times the number of threads.

algorithm, the duration of executing a task code is so tiny that the steal decision step becomes more critical. A way to reduce the overhead might be to steal more than one task. It would nevertheless require a study to determine the amount of tasks to steal, like in libKOMP or in the OpenMP runtime developed over the Rose compiler. As for libGOMP, there are so many accesses to the global list during the whole run that it didn't finished in a reasonnable duration.

4.6 Sort

The *Sort* benchmark sorts a random permutation of n numbers with a fast parallel sorting variation of the classical mergesort. As seen in Figure 4, the best speed-up for MPCOMP corresponds to the *one list per thread* granularity (first parameter). Regarding the second parameter (stealing strategy), the *Random* policy is the most performant. This can be explained by the fact that this application is well balanced with nested tasks. Therefore, the strategy with the lowest overhead performs better. This is confirmed by the Intel OpenMP performance which goes a little further probably because of some additional runtime optimizations.

4.7 SparseLU

The Sparse LU application computes a LU matrix factorization over sparse matrices. We focus on the multiple producer version. A group of submatrices is assigned to each thread and, due to the sparseness of the matrix, some of them may not be allocated which explains the unbalance of the algorithm. Intel OpenMP performs really well here with an execution time of 25.52 seconds with 128 threads whereas libGOMP does not finish in a reasonable time. As for MPCOMP strategies, the one using one list per motherboard and a hierarchical stealing is the best approach with a speedup equal to 27 corresponding to a run

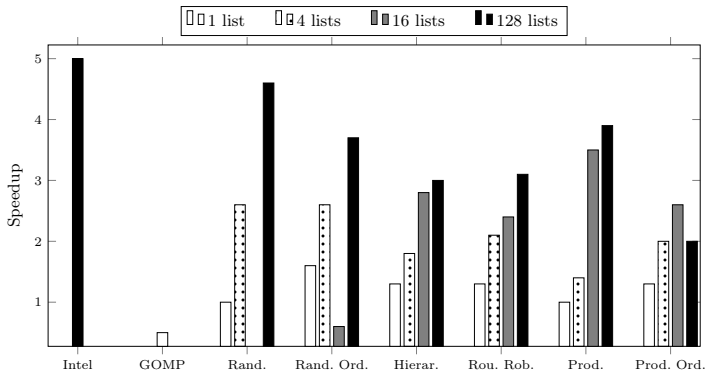

Fig. 4. Speedup for *Sort* benchmark with 128 threads

of 32.78 seconds. Even though the speedup of Intel OpenMP is far better (near 110), one has to notice that it comes from a long serial execution time for Intel OpenMP, equal to 2 993 seconds, in comparison to libGOMP with 1 005 seconds or to MPCOMP with 985 seconds.

Thus, a work-stealing strategy that takes care of data locality is the recommended one. Indeed, this benchmark presents a large percentage of accesses to shared data. Moreover, unlike the previous algorithms which employ nesting task creation, those sharings do not happen between tasks and their children. So there is less likely for an OpenMP thread to get tasks that will work on the same data.

4.8 Analysis

This experimental section shows that the two parameters highlighted in this paper for task scheduling are relevant. For the first parameter (task-list granularity), one list per core seems to be the best suitable configuration in most cases. Even if this result might be surprising on such manycore architecture (2 NUMA levels and 16 sockets), allocating one specific list per OpenMP thread in the team allows to reach decent performance. However, this study does not focus on memory consumption, but this aspect might become important on future generations of manycore architectures. In such situation, the solution of one list per thread may not be possible anymore depending on the amount of memory used by the application data.

Regarding the second parameter (work-stealing strategy), the performance really depends on the characteristics of the benchmark. Even if our implementation of each strategy can be optimized, the overhead to decide the target list to steal is crucial. For example, in the Fibonacci benchmark, each task is tiny. It explains why a low-overhead strategy, like random, performs well. Nevertheless, the round robin approach is able to reach high performance in such cases because it allows an approximate hierarchical traversal with a lower overhead. For benchmarks with tasks containing a larger amount of work, strategies with a higher overhead (hierarchical and producer) can be considered. The hierarchical approach allows to select the target list with respect to the data locality: this is an important parameters on such manycore architectures.

5 Conclusion and Future Work

High-Performance Computing platforms are getting embarassingly parallel for application developers. They need to split up their algorithm structure to offer enough parallel work to the massive number of cores forming the current and future architectures. The evolution of the parallel programming models, like OpenMP with the task support, allows them to express their problem in a finer-grained parallelism. Nevertheless, the hierarchical structure of the underlying system is barely considered in most of the OpenMP task schedulers. On large NUMA nodes it can be really penalizing due to the overhead of distant

memory accesses over local ones. We proposed in this paper an evaluation of two key parameters for task scheduling. We designed a configurable task scheduler which allows us to control: (i) the granularity of task list, according to the hardware topology, and (ii) the work-stealing strategy. Thus, we compared those technic combinations through the execution of *Barcelona OpenMP Task Suite*. We noticed that the approach chosen by the Intel OpenMP Runtime Library is suitable for the first parameter (list granularity) and sometimes for the second too. However the design adopted by LibGOMP cannot scale on a large NUMA node for several applications. Finally, we show that hierarchical work-stealing strategies can lead to better performance in specific cases.

For future work, this study would benefit from using the same compiler to avoid performance difference for the computational part of the applications. Thus, we are actually working on the support of both Intel C compiler and recent GCC compiler in the MPC framework for further fair comparisons. Regarding the current key parameters, our approaches may be optimized and tested on more applications to derive a static/dynamic cost model to choose the right strategy. Finally, some new parameters could be taken into account to refine this study. For example, the cut-off value can be used to artificially stop the parallelism extraction and reduce the runtime overhead. The memory consumption is another aspect that could be of interest for a task runtime. Indeed, this metric may become more important on next-generation of manycore processors (e.g., Intel *Xeon Phi*) with an hybrid programming model like MPI+OpenMP.

References

1. Ma, T., Bosilca, G., Bouteiller, A., Goglin, B., Squyres, J.M., Dongarra, J.J.: Kernel Assisted Collective Intra-node MPI Communication among Multi-Core and Many-Core CPUs. In: International Conference on Parallel Processing (ICPP), pp. 532–541 (2011)
2. Broquedis, F., Furmento, N., Goglin, B., Wacrenier, P.-A., Namyst, R.: ForestGOMP: An efficient OpenMP environment for NUMA architectures. International Journal on Parallel Programming, 418–439 (2010)
3. Jin, H., Jespersen, D., Mehrotra, P., Biswas, R., Huang, L., Chapman, B.: High performance computing using MPI and OpenMP on multi-core parallel systems. Parallel Computing, 562–575 (2011)
4. The OpenMP API specification for parallel programming, `http://www.openmp.org`
5. An OpenMP implementation for GCC, `http://gcc.gnu.org/projects/gomp`
6. Intel Xeon Phi Coprocessor - The Architecture. `http://software.intel.com/en-us/articles/intel-xeon-phi-coprocessor-codename-knights-corner`
7. Intel OpenMP Runtime Library, `https://www.openmprtl.org`
8. Terboven, C., Schmidl, D., Cramer, T., an Mey, D.: Assessing OpenMP Tasking Implementations on NUMA Architectures. In: Chapman, B.M., Massaioli, F., Müller, M.S., Rorro, M. (eds.) IWOMP 2012. LNCS, vol. 7312, pp. 182–195. Springer, Heidelberg (2012)
9. Pérache, M., Jourdren, H., Namyst, R.: MPC: A unified parallel runtime for clusters of NUMA machines. In: Luque, E., Margalef, T., Benítez, D. (eds.) Euro-Par 2008. LNCS, vol. 5168, pp. 78–88. Springer, Heidelberg (2008)

10. Addison, C., LaGrone, J., Huang, L., Chapman, B.: OpenMP 3.0 tasking implementation in OpenUH. Open64 Workshop at CGO (2009)
11. Blumofe, R.D., Joerg, C.F., Kuszmaul, B.C., Leiserson, C.E., Randall, K.H., Zhou, Y.: Cilk: An Efficient Multithreaded Runtime System. Journal of Parallel and Distributed Computing, 207–216 (1995)
12. Liao, C., Quinlan, D.J., Panas, T., de Supinski, B.R.: A ROSE-Based OpenMP 3.0 research compiler supporting multiple runtime libraries. In: Sato, M., Hanawa, T., Müller, M.S., Chapman, B.M., de Supinski, B.R. (eds.) IWOMP 2010. LNCS, vol. 6132, pp. 15–28. Springer, Heidelberg (2010)
13. Olivier, S., Porterfield, A., Wheeler, K.B., Spiegel, M., Prins, J.F.: OpenMP task scheduling strategies for multicore NUMA systems. International Journal of High Performance Computing Applications, 110–124 (2012)
14. Wheeler, K.B., Murphy, R.C., Thain, D.: Qthreads: An API for programming with millions of lightweight threads. In: IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2008, pp. 1–8 (2008)
15. Gautier, T., Ferreira Lima, J.V., Maillard, N., Raffin, B.: XKaapi: A Runtime System for Data-Flow Task Programming on Heterogeneous Architectures. In: IEEE International Parallel and Distributed Processing Symposium (IPDPS), pp. 1299–1308 (2013)
16. Broquedis, F., Gautier, T., Danjean, V.: LIBKOMP, an Efficient OpenMP Runtime System for Both Fork-Join and Data Flow Paradigms. In: Chapman, B.M., Massaioli, F., Müller, M.S., Rorro, M. (eds.) IWOMP 2012. LNCS, vol. 7312, pp. 102–115. Springer, Heidelberg (2012)
17. Agathos, S.N., Kallimanis, N.D., Dimakopoulos, V.V.: Speeding up OpenMP tasking. In: Kaklamanis, C., Papatheodorou, T., Spirakis, P.G. (eds.) Euro-Par 2012. LNCS, vol. 7484, pp. 650–661. Springer, Heidelberg (2012)
18. Augonnet, C., Thibault, S., Namyst, R., Wacrenier, P.-A.: StarPU: A unified platform for task scheduling on heterogeneous multicore architectures. In: Sips, H., Epema, D., Lin, H.-X. (eds.) Euro-Par 2009. LNCS, vol. 5704, pp. 863–874. Springer, Heidelberg (2009)
19. Broquedis, F., Clet-Ortega, J., Moreaud, S., Furmento, N., Goglin, B., Mercier, G., Thibault, S., Namyst, R.: Hwloc: a Generic Framework for Managing Hardware Affinities in HPC Applications. In: The 18th Euromicro International Conference on Parallel, Distributed and Network-Based Computing, PDP, pp. 180–186 (2010)
20. Duran, A., Teruel, X., Ferrer, R., Martorell, X., Ayguade, E.: Barcelona OpenMP Tasks Suite: A Set of Benchmarks Targeting the Exploitation of Task Parallelism in OpenMP. In: Proceedings of the 2009 International Conference on Parallel Processing, pp. 124–131 (2009)
21. Mahéo, A., Koliaï, S., Carribault, P., Pérache, M., Jalby, W.: Adaptive OpenMP for Large NUMA Nodes. In: Chapman, B.M., Massaioli, F., Müller, M.S., Rorro, M. (eds.) IWOMP 2012. LNCS, vol. 7312, pp. 254–257. Springer, Heidelberg (2012)